Model-Driven Engineering for Software Migration in a Large Industrial Context

Franck Fleurey[1,2], Erwan Breton[2], Benoit Baudry[1], Alain Nicolas[2], and Jean-Marc Jézéquel[1]

[1] IRISA/INRIA, Rennes, France
{ffleurey, bbaudry, jezequel}@irisa.fr
[2] Sodifrance, Nantes, France
{ebreton, anicolas}@sodifrance.fr

Abstract. As development techniques, paradigms and platforms evolve far more quickly than domain applications, software modernization and migration, is a constant challenge to software engineers. For more than ten years now, the Sodifrance company has been intensively using Model-Driven Engineering (MDE) for both development and migration projects. In this paper we report on the use of MDE as an efficient, flexible and reliable approach for a migration process (reverse-engineering, transformation and code generation). Moreover, we discuss how MDE is economically profitable and is cost-effective over the migration through out-sourced manual re-development. The paper is illustrated with the migration of a large-scale banking system from Mainframe to J2EE.

1 Introduction

Positioned from the mid 80's on IT servicies dedicated to Banks and Insurance Companies, Sodifrance has developed a strong legacy modernization expertise based on software solutions to industrialize transformation projects. Since 1994, Sodifrance has adopted and promoted model-driven engineering (MDE) approaches for modernization projects. It has industrialized model-driven techniques for reverse-engineering, code analysis and transformation and for representing and manipulating information systems. These solutions allow the company to propose efficient and profitable solutions for migration and modernization of software legacy systems.

In this paper we first present an original model-driven process, developed at Sodifrance, for software migration. This process includes automatic analysis of the existing code, reverse engineering of abstract high-level models, model transformation to target platform models and code generation. We detail the different meta-models and transformations that are produced for the automation of these steps. We also discuss what artefacts can be directly reused and which ones need to be adapted from one project to another. Sodifrance has developed a tool suite for model manipulation called Model-In-Action (MIA) that is used as a basis for automating the migration.

G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735, pp. 482–497, 2007.

A second contribution of this paper is an industrial feedback on the benefits and issues of MDE for migration. First, we present data for a migration project of a large-scale banking system from Mainframe to J2EE. These data are used to discuss the improvements with respect to efficiency, flexibility and reliability that are introduced with a model-driven solution migration. Moreover, we compare MDE and complete manual re-development, and discuss how MDE is economically profitable and is cost-effective.

2 Model-Driven Migration Process

The constant evolution of software technology leads to continuous migrations of software components. These projects may be motivated by different reasons such as the obsolescence of a technology, the pressure of users, or the need to build a single coherant information system when merging companies. Most of the time software migration is achieved through the full re-developement of the legacy application. Model-driven software developement offers an oportunity for increasing the automation in software migration.

The full automation of migration is difficult to achieve not only because of the distance between the legacy platform and the new platform but also in order to ensure the quality of the new application. Most of the time, the objective of migration is not to simply "compile" the legacy application to a new platform but to create a new version of the application using state of the art development techniques. This is necessary to ensure the maintainability of the new application and to leverage the latest technologies in terms of graphical user interfaces, distribution and mobility.

In the following, section 2.1 first presents the general process developed by Sodifrance for model-driven migration, section 2.2 discusses the automation of the process and section 2.3 details how this process is adapted in practice along the phases of a migration project.

2.1 Migration General Process

Figure 1 presents the general process developed by Sodifrance for model-driven migration. This process is mainly divided in four steps.

The first step is the parsing of the code of the legacy application, to build a complete model of the code of the application. This step can be divided into two stages: first a parser builds an abstract syntax tree from the code and, then this syntax tree is processed by a transformation to build an actual model that conforms to the meta-model of the legacy language. During the second stage, all the symbols such as types, variables or function calls are resolved and properly bound to the appropriate model elements. This is a necessary step to allow for a efficient analysis of the legacy system. The meta-model denoted L on figure 1 corresponds to the meta-model of the legacy application implementation language.

The second step is a reverse-engineering from the code model to a platform independent model. The role of this step is to abstract high-level views from the

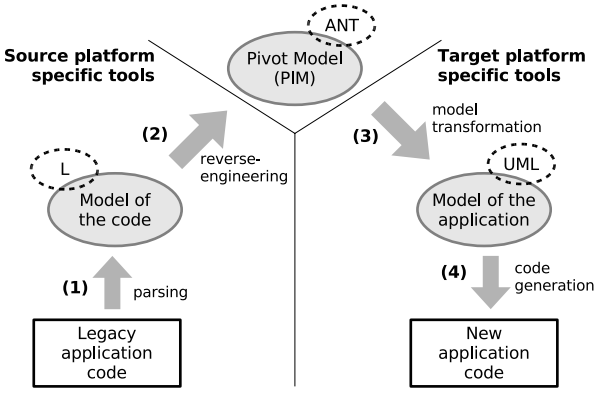

Fig. 1. Model-driven migration priciple

model of the code. This step is implemented by model transformations from the legacy language meta-model (L) to a pivot meta-model. The pivot meta-model used by Sodifrance is a platform independent meta-model called ANT which contains packages to represent:

- Static data structures (close to the UML class diagram).
- Actions and algorithms (it includes an imperative action language).
- Graphical user interfaces and widgets.
- Application navigation.

The navigation is the most high level view of the ANT meta-model. Figure 2 shows an excerpt of this meta-model. It connects dialog elements which correspond to GUI forms, transitions between forms and their GUI events with operations in the class model.

All ANT views have to be created through model transformations from the model of the code of the legacy application. In order to be able to create high-level views, such as a model of the graphical user interface of the legacy application, the model transformations have to rely on a knowledge of the libraries of the legacy platform and on coding conventions (or code patterns introduced by tools) that were used during the development of the legacy application. This is the reason why, even if the legacy platforms for several migration projects are similar, the legacy code must be carefully studied in order to properly adapt the migration tools to every single project.

The third step is the transformation of the ANT model into a platform specific model of the application. This step is implemented using model transformations from the ANT meta-model to the UML meta-model. These transformations are design transformations which refine the platform independant views of the pivot model to fit the target platform. Again at this stage, it is important to adapt the transformation to meet the requirements of every customer. This issue is discussed with more details and illustrated on a specific project in section 4.

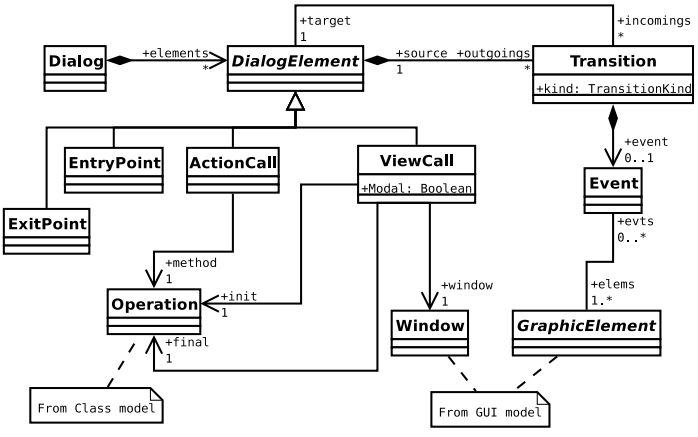

Fig. 2. Excerpt of the ANT navigation meta-model

The last step is the generation of the code of the new application from the platform specific model. To implement this step, Sodifrance uses template-based text generation tools in order to be able to easily customize code generation acording to the customers requirements. The specific tools used by Sodifrance for the implementation of model-transformations and code generation are presented section 3.

2.2 Automation in the Migration Process

To reduce the cost of migration the goal is to achieve an optimum automation in the migration process. However, this should not impact the quality in terms of design, performances or maintainability of the resulting application. Since the legacy application is fully-executable and the target platform is usually powerful enough, one could argue that the migration should be completely automated. It is theoretically possible: it would be the equivalent of writing a compiler for the legacy language that targets the new platform.

However, as stated in the previous section, migration, and especially in the context of modernization, is more than just creating an executable version of the application on top of the new platform. The goal is to design the application for the new platform in order to make it more efficient, more reliable, easier to maintain or easier to extend than the legacy application. In practice this means that the new code should respect the coding standards and best practices of the target platform languages, it should take into account the specific requirements related to the software development process used by the customer company, there should be models for the new application, etc.

In the migration process implemented by Sodifrance the first two steps (as presented on figure 1) are usually completely automated, i.e. all the information from the legacy system is represented in the pivot model. This is to concentrate the manual effort on the transformation from the pivot model to the new

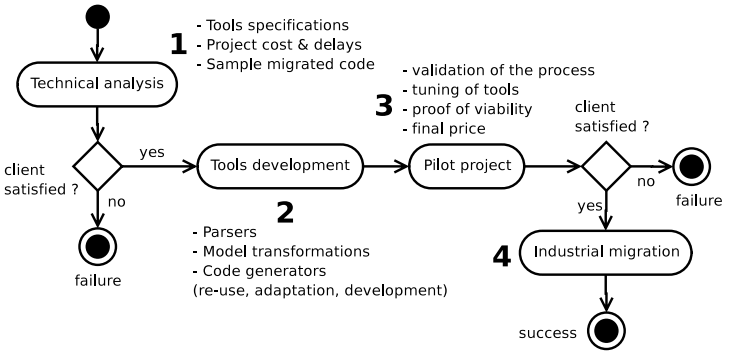

Fig. 3. Model-driven migration project phases

application and avoid having to deal manually with the legacy code as a whole. If some elements of the legacy code cannot fit properly in the pivot model, these elements are captured as notes or tags and presented to the developer when the corresponding parts of the application are transformed or generated.

To maximize the efficiency of the migration process, the tasks that are left to the developer have to be clearly identified and the developer should be provided with all the information he or she needs. This is taken into account in the design of the transformations and code generators. For example in the case of a Java code generator, *TODO* directives can be generated for every piece of code that requires manual inspection, re-factoring or completion. This *TODO* directive can contain the kind of work that has to be done and references to the model elements that are relevant to it. The *TODO* directives are summarized into a task list which gives the developer a clear view of what has to be done.

2.3 Migration Project Phases

Prior to the actual migration and implementation of the new application, the design, the implementation and the validation of a project specific migration process must be completed. This includes the parsing of legacy languages, reverse engineering transformations, high-level design of the new application and mappings between the structures of the legacy application and the concepts of the target platform. All these tasks require some effort due to their complexity and their overall influence on the migration project. In the project structure used by Sodifrance, as represented on figure 3, there are three project phases before the actual migration can start.

The first phase represented on figure 3 is a technical analysis. Its objective is to study the legacy platform, define the target platform and specify the tools that are needed by the migration process. This phase is crucial for the migration project. It is used to estimate the effort that would be required for the development of the tools and the total effort that would be required for the migration. At the end of the technical study a total contractual price is proposed to the customer. During the technical study a small component of the legacy application

is usually migrated using generic tools and manually completed to match the code that would be produced using the final tools. This serves as a test for the tool specifications and as a demonstration of the resulting code the customer can expect. If both the price proposed by Sodifrance and the quality of the migrated code are satisfactory to the customer, the project can carry on.

The second phase represented on figure 3 is a tool development phase. The objective is to develop all the tools that have been specified for the migration process. Most of the time the tools do not have to be developed from scratch but are rather re-used or adapted from previous projects. However, most of the time even if the language is the same, the language version and the coding style might be different and require some adaptation.

The third phase represented on figure 3 is a pilot project. The objective of the pilot project is to validate and fine tune the migration process and the tools it uses. It also serves as a demonstration of the viability of the process and allows measuring its efficiency precisely. During this phase, a component of the legacy application is used as a benchmark for the migration process. This component has to be chosen to be as representative as possible of the components of legacy application. In practice the development of the pilot project is truly a testing and debugging phase for the migration tools. For this reason it is usually a lot longer than the migration of a comparable component once the migration process is fully-functional. At the end of the pilot project, the customer is provided with a final price for the project and has a sample of how the new application would look like.

Projects seldom have to stop after the pilot project: the actual migration usually starts shortly afterwards. The preparation of a model-driven migration process can be quite long (the three phases described previously usually require around 6 months to complete but can last up to a year on specific projects such as the one described in section 4), but once the process is up and running, the migration rate can be far more rapid than with any competing techniques. This is discussed in section 5, but before that, the next section presents the model-driven engineering tools used by Sodifrance to practically implement model-driven migration.

3 Model-In-Action (MIA) Tool Suite

Implementing the migration process presented in the previous section requires advanced, scalable and reliable tools for model transformation and code generation. For both the needs of migration project and development projects, Sodifrance has developed Model-In-Action (MIA) [1], a suite of model-driven engineering tools. This section gives a quick overview of these tools.

Figure 4 presents a simplified architecture diagram for the MIA tools. One of the essential requirement for a company like Sodifrance is to be able to adapt to any specific modeling technology used by their clients. In the design of MIA this has been taken into account by creating a generic modeling platform that can connect through various drivers to existing repositories and modelers. On

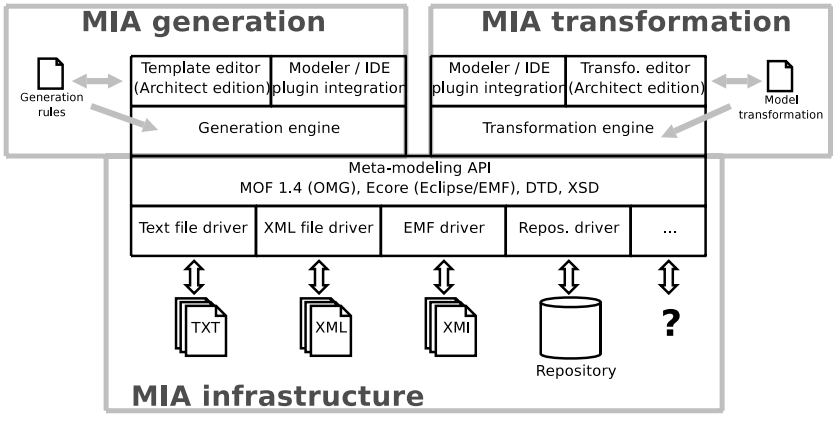

Fig. 4. Model-In-Action tool suite architecture

top of this generic modeling layer the suite is composed of two main products: *MIA-Transformation* for model-to-model transformation and *MIA-Generation* for code generation. Each of these tools is divided in three types of components:

- Core engines for model transformations and code generation. These components are on top of the meta-modeling API and do not have any user interface. They are responsible for the execution of model transformations and code generators.
- Development environments for model transformations and code generators (MIA Architect environments). These environments are used by software architects to design and implement the model transformations and code generators required by MDE projects.
- User environments for model transformation and code generators (MIA developer environments). There are not only standalone versions of these tools but also plug-in versions that integrate directly in the IDEs and modelers of the software developers.

MIA-Transformation is a rule-based model-to-model transformation engine. A model transformation is defined by a set of rules defined between some input meta-models and some output meta-models. Each rule is composed of three elements:

- A context: it corresponds to the set of declared variables and parameters.
- A query: it is an expression that calculates the set of model elements to be processed by the rule.
- An action: it can be a creation, a modification or a deletion of model elements and is performed for each model element returned by the query.

When using MIA-Transformation, alternative languages may be used for expressing transformation rules. MIA-Transformation includes both a fully declarative language (close to the declarative form of QVT) and an imperative language.

The two languages can even be mixed in a single transformation rule: the query can be written using the declarative language and the action implemented imperatively. In addition, as rule based transformation has some limitations, it is possible to define transformation services in Java and use them in transformation rules.

MIA-Generation is a template based model-to-text transformation engine. The idea of MIA-Generation is to attach text generation scripts directly in meta-models in order to define how each model element should be generated. There are two kinds of scripts:

- Templates that textually describe the piece of code to be generated.
- Macros that allow more complex operations such as string handling or model navigation.

The macros are defined directly in Java and can be called from the template. The fact that the generation scripts are directly attached to the meta-model makes MIA text generators easy to understand, adapt and maintain. In addition, the generation engine can keep track of the execution of each generation script and the text it has produced. This provides the developer with all the information required to tune or fix a code generator.

4 Migration of a Large-Scale Banking Application

This section reports on how the migration process described in section 2 is applied in the context of a large-scale banking application. The migration of this application is part of the modernization of the information systems of a French bank[1]. The objective of the project was to migrate a mainframe system made of around a million lines of code to J2EE in order to ease the maintenance and future evolutions of the system. The overall system is composed of:

- 42 applications (for a total of 800 forms and 7500 events)
- 99 prints and exports using Cristal Report
- 990 server services
- 20 batch processes

Sodifrance (and their model-driven migration approach) was chosen by the bank for the migration of this system not only because of the quality assurance provided by the use of automation but also for pricing reasons. After an initial study of the project by Sodifrance and several competing companies, the price proposed by Sodifrance was significantly lower than the price of any brute-force redevelopment strategy (out-sourced or not). In the following, section 4.1 presents the customer's requirements and the migration process that has been developed, section 4.2 details the project schedule and section 4.3 discusses the problem of the validation of the migrated application.

[1] For confidentiality reasons, and for the protection of Sodifrance customers, this paper does not provide specific details on the migrated application.

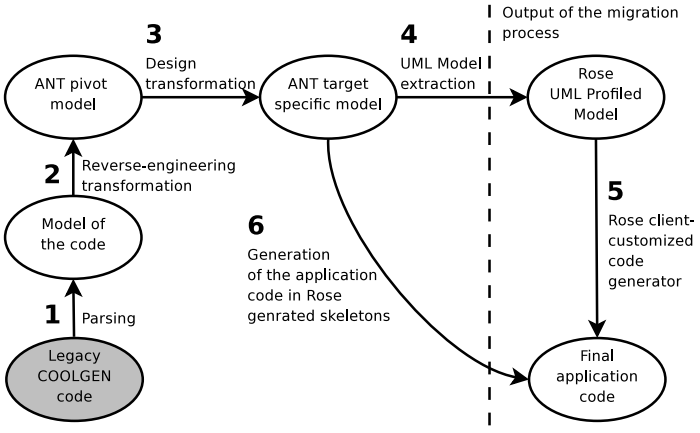

Fig. 5. Banking application migration process

4.1 Specific Requirements and Migration Process

For the modernization of its information system both the servers and the client applications of the bank had to be migrated. The whole legacy application had been developed using the COOLGEN IDE. COOLGEN provides an intermediate programming language and produces executable application by compiling this language to a combination of C code and COBOL code. For the modernization of the system, the servers had to be migrated to plain COBOL because the code generated by COOLGEN was difficult to maintain and had some performances issues. The 42 client applications had to be migrated from COOLGEN to J2EE web applications. The applications and the servers would communicate through a COBOL/Java middleware. The following focuses on the migration of the 42 client applications from COOLGEN to J2EE.

An important requirement of the customer for this project was the strict respect of its internal development standards. All the new applications developed by this bank are generated from Rational Rose UML models. All the models conform to a UML profile developed by the bank itself and specific code generators are used. As a result of the migration process the bank expected to be able to round-trip between models and code using its usual profiles, tools and code generators. The model-driven migration process had to be adapted to take this specific requirement into account.

Figure 5 presents the migration process that is applied to each of the 42 applications of the legacy system. Steps 1 and 2, which correspond to the parsing and reverse-engineering of the application, are similar to the two first phases of the general process presented in section 2. These two phases produce an ANT model of the legacy application which includes all the information contained in the code of the applications (windows, widgets, statements, expressions). Step 3 (also quite similar to the third step of the general process) does the mapping

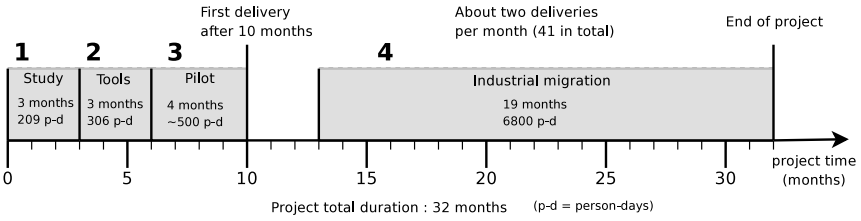

Fig. 6. Banking application migration time schedule and cost breakdown

between the source architectural concepts and the target ones to produce a complete platform specific model of the target application.

Steps 4, 5, and 6 of the process presented figure 5 are specific to the banking system migration and designed to produce customer-specific synchronized UML models and source code for the target application. Firstly, step 4 is a model transformation that extracts a UML-profiled model from the ANT application model. The elements of the target application, such as statements, that do not fit in the UML-profiled model are ignored. Then, step 5 consists in using the regular code generator used in all the development projects of the bank to produce code skeletons from the UML model. In regular projects these skeletons have to be filled manually but here the role of step 6 is to automatically generate the final application code in the code skeletons. The manual phase of the migration can then be carried out: the model transformations and the code generators have left notes in the UML model and comments in the code wherever a manual migration task has to be accomplished.

4.2 Project Time Schedule and Cost Breakdown

This section details the organization and cost breakdown for the banking system migration. The overall project required a total of 9315 days of work including 7815 for the migration of the 42 client applications. As discussed in section 2 any model-driven migration project has several mandatory initialization phases to design a specific migration process and adapt or develop the required tools. Figure 6 presents the scheduling and the cost (in terms of days of work) for each phase of the banking system migration project.

The first preliminary phase of the project is the technical study. In the case of the banking system it took 3 months and required a total of 209 days of work (which represents about 2.5% of the project effort). Then, the tool development phase and the pilot project took 7 months to complete and required an approximate effort of 800 days of work (around 10% of the project effort). For the pilot project, a representative client application has been chosen among the 42 application that had to be migrated. The delivery of the pilot project occurred 10 months after the beginning of the project and after about 12% of the project effort has been spent.

The important investment and delay before the first delivery is specific to model-driven migration. Moreover, because the preliminary tasks are difficult to

parallelize and because the developers need to have a global view of the project to accomplish these tasks, using a large team of developers cannot really help reducing the duration of preliminary work. The developer team for these tasks have to be small (3 to 8 developers for most Sodifrance projects) and should include experts of the source platform, experts of the target platform and model transformation experts.

On the banking application the industrial migration of the 41 remaining application started 3 months after the end of the pilot project. This phase required a total of 19 months to complete. During this industrial phase of the project the migration is performed in parallel by three independent teams of around 15 developers each. Sodifrance migrates three applications at a time, and, during the 19 month period of the industrial migration an average of 2 deliveries are made per month. Contrary to the project preliminary phases that require a small developer team, the industrial migration duration can easily be shortened by increasing the number of developers.

4.3 Validation and Quality Assurance

Even with the use of automation, since there is still a significant part of the work done by hand, the migrated application has to be carefully validated in order to check its correctness, performance and integration in its new environment. In practice this is achieved thanks to a strict non-regression testing process. This test process is costly for the customers because they have to provide test cases together with the legacy applications and they have to perform acceptance testing[2]. It is also costly for Sodifrance who perform unit testing for the new application and uses the test cases provided with the legacy application to do regression testing. In the case of the banking application the total testing cost is around 3500 days of work (around 1000 days for unit testing and 2500 for regression testing). This represents 45% of the total project cost.

5 Discussion

This section compares *model-driven migration* with brute-force *re-development* migration strategies. The most significant difference between the two approches is the significative *preliminary tasks* required by model-driven techniques. This section especially discusses the influence of these preliminary tasks on the schedule and cost breakdown of migration projects and shows that for projects over a critical size, the model-driven approach is more profitable than re-development.

Complete re-development has some advantages over automated migration. Firstly the development process is similar to the development of any application except that it has a fixed and non-ambiguous specification. This allows using efficient software engineering techniques which is reassuring and unsurprising

[2] The numbers provided in this section do not include this cost. Only the cost for Sodifrance is taken into account.

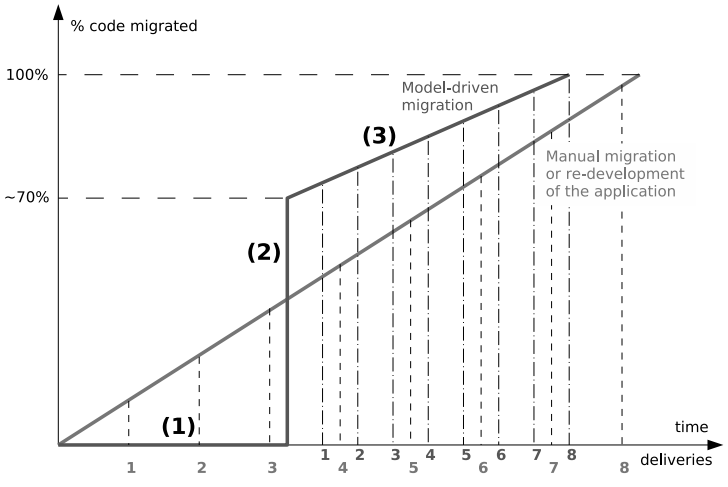

Fig. 7. Migrated code percentage in function of time

to the customer. Secondly, the target application can easily be re-designed, re-factored and adapted to the new platform. Thirdly, evolutions to the legacy application can be taken into account in the design of the new application. All these advantages, most of the time combined with out-sourcing to cut workforce cost, allow full re-development to be a common option for modernization.

In this context, thanks to model-driven migration, Sodifrance has managed to provide a comparable quality of service at lower prices to its customers on a number of modernization projects.

5.1 Migration Time Schedule

Figure 7 compares re-developpment and model-driven migration with respect to the percentage of code migrated over time. For re-development the model we use is linear: the components of the legacy application are re-developed one by one. For model-driven migration the process is a little different: during the first stage of the project (1) the objective is to develop the tools that will be used to partially automate the migration. During this first stage no code of the new application is produced at all but once the tools are fully functional they typically allow generating about 70 percent of the final application code (2). The actual migration can then begin (3), each component of the legacy application is manually completed and delivered to the customer.

The most important difference between the two approaches is the first phase of the model-based process which is an investment in specific tools that will make the migration faster. One of the drawback of model-driven migration is that for an initial period of time, no final code is produced and thus nothing can be delivered to the customer. In the example of figure 7 the legacy application has been divided in 8 components. Using a re-development strategy, the first

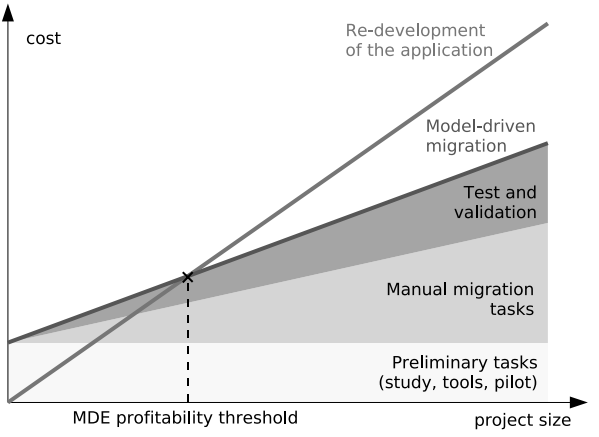

Fig. 8. Project cost in function of it size

component can be delivered to the customer just after the beginning of the project. On one the hand the first component is delivered after quite a long period of time: using model-driven migration, when the first component is delivered, 3 components have already been finished with re-development. But on the other hand, using model-driven migration, once the production of the new application has started, the delivery rate can be faster than for re-development. Eventually, the delivery of components developed using the model-driven approach can catch up with the delivery of re-developed components (this is the case for components 7 and 8 on the figure).

In the case of the banking application described previously, the preliminary tasks of the migration project required 10 months which represents about a third of the total project duration. From an economical point of view, more than 10% of the total migration cost was spent on these preliminary tasks. The next subsection discusses the profitability of this investment.

5.2 Migration Cost Repartition

To be profitable, the model-driven migration process must be applied on legacy applications that have a sufficient size. Indeed, the effort that has to be invested for developing migration tools mostly depends on the complexity of the input and output platforms but not on the volume of code that has to be migrated. Figure 8 presents an estimation of the cost of a migration project using both model-driven migration and re-development.

In the case of re-development the cost is directly proportional to the size of the legacy application. In the case of model-base migration, there is a fixed initial cost related to the development of tools which is complemented by a linear cost corresponding to manual migration efforts. The gradient of the function corresponding to model-driven migration is lower than the gradient for

re-development because using migration tools reduces the manual effort that has to be provided.

A general profitability threshold for model-driven migration cannot be estimated accuratly because it really depends on the ratio of tools that have to be developed for each project. In practice, the experience of Sodifrance on model-driven migration shows that the profitability threshold for MDE in the context of migration is quite low. Even for projects that require about 1000 days of work, the initial overhead of developing tools pays off. On the migration of the large banking application described previously, Sodifrance estimated that the cost of re-development whould have been around twice the price of model-driven migration.

5.3 Benefits and Limitation of Model-Driven Migration

The primary advantage of model-driven migration is to partly automate the migration process. As discussed in the previous sections this allows Sodifrance to significantly lower the prices and duration of migration projects. This is the reason why Sodifrance is often chosen over competing companies that propose full re-development.

The second advantage is to allow for reuse between migration project. This is another element that allows cutting the cost of migration. All the transformations and tools that have been developed for a migration projects can be adapted to future project that have similar input or output platforms.

The first limitation of model-driven migration is a commercial limitation related to the cost and time consumed by preliminary tasks. In the project presented in section 4 the first delivery of migrated code occured after 10 months. This is a commercial issue because after the begining of the project the customer has to wait for a long time without seeing the progression of the project. To mitigate this issue, possible solution is to works in close collaboration with the IT department of the customer and, if possible, to include members of the customer company in the development team.

The second limitation is related to the cost of testing. This is not specific to model-driven migration but is a general problem in software migration. For the banking application discussed previously, testing represents 45% of the total migration cost. This cost does not include the cost of the production of regression tests and acceptance testing which are the responsability of the customer [3]. One of the reasons of the important cost of testing tasks is that, for most migration project, they are mostly handled manually. In the same way model-based techniques has been applied to smartly automate repetitives migration tasks, Sodifrance is now studying model-based regression testing using meta-modeling languages such as Kermeta [2] to reduce the cost of testing.

[3] The production of the tests is a cost that has to be taken into account by the client. However, this cost is usually far lower than the cost of providing the complete specifications required for full re-development.

6 Related Works

Software modernizaton has been identified by the OMG as an important application field for model-driven architecture. The Architecture-Driven Modernization (ADM) is an OMG task force dedicated to this topic [3] that aims at building standard metamodels and tools for software modernization. Reus et al. in [4] propose a MDA process for software migration that is quite similar to ours. They parse the text of the original system and build a model of the abstract syntax tree. This model is then tranformed into a pivot language that can be translated into UML. A prototype automates parts of this process using ArcStyler [5]. Bordbar et al. [6] propose a model-based approach for maintenance of data-centric systems. Their MDA approach improves the evolution and maintenance of databases in applications developed with java and modelled with UML. In [7], Zou transforms legacy code towards object-oriented languages. Parts of this process are implemented with automatic program transformations.

Another type of works related to the study presented in this paper concerns feedbacks from industrial projects that have applied model-driven approaches. In the last two editions of the MoDELS conference, two studies gave such feedback. In [8], Baker et al report on the significant improvements in productivity and reliability gained with MDE techniques and also present the remaining issues to profit more from those approaches. In [9], Staron presents a study on the requirements for the adoption of MDE in software industries. The paper reports on the observations of two companies that tried to MDE in their development process. In [10] M1_Global_solution compares MDE and off-shore development. They conclude that MDE tool increased developer productivity by over 50 percent and advocate a combination of MDE for automatic production of a large part of the system and off-shore development for the parts that need to be manually developed.

7 Conclusion

This paper has precisely presented a model-driven migration and modernization process developed by the Sodifrance company. We have detailed the process and the tools that automate this process. The paper has also dicussed the benefits introduced by MDE in terms of reuse and automation, and also the issues that are introduced to fully benefit from reusable transformations and generators. Finally we have showed that, even if the process is not fully automated and requires manual adaptation from one project to the other as well as manual implementation of some parts of the final application, it is still viable compared to manual re-development.

Even if model-driven engineering is already economically profitable for migration, there are still some important challenges that need to be tackled. A major issue in terms of human effort is testing. Today, regression test is used to validate the migration. However, the production of efficient regression test cases is manual, ad-hoc and difficult to evaluate. Future work consists in adding, in the

reverse-engineering phase, a step to reverse a model for high-level control flow in the application in order to eveluate test coverage at use-case level. Moreover, unit and integration test for the migrated code is also very expensive. A possible solution here could consist in generating test objectives when generating the code.

References

1. Sodifrance: Model-in-action tool-suite (2007), http://www.mia-software.com/
2. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: MODELS/UML 2005, Montego Bay, Jamaica. LNCS, Springer, Heidelberg (2005), http://www.kermeta.org/
3. OMG: Architecture-driven modernization (2006)
4. Reus, T., Geers, H., Deursen, A.v.: Harvesting software systems for mda-based reengineering. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 213–225. Springer, Heidelberg (2006)
5. Objects, I.: Arcstyler (2007)
6. Bordbar, B., Draheim, D., Horn, M., Schulz, I., Weber, G.: Integrated model-based software development, data access, and data migration. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 382–396. Springer, Heidelberg (2005)
7. Zou, Y., Kontogiannis, K.: Migration to object oriented platforms: a state transformation approach. In: ICSM'02 (International Conference on Software Maintenance), pp. 530–539 (2002)
8. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context - motorola case study. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)
9. Staron, M.: Adopting model driven software development in industry - a case study at two companies. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 57–72. Springer, Heidelberg (2006)
10. M1 Global Solutions: Model driven software development and offshore outsourcing (2004)